MPI-IO In-Memory Storage with the Kove XPD

Julian Kunkel
Deutsches Klimarechenzentrum
Hamburg, 20146
Email: kunkel@dkrzde

Eugen Betke
Deutsches Klimarechenzentrum
Hamburg, 20146
Email: betke@dkrz.de

***Abstract*—Many scientific applications are limited by the performance offered by parallel file systems. SSD based burst buffers provide significant better performance than HDD backed storage but at the expense of capacity. Clearly, achieving wire-speed of the interconnect and predictable low latency I/O is the holy grail of storage. Throughput and latency of in-memory storage promises to provide optimal performance that exceeds performance of SSD based solutions. Kove®'s XPD® offers pooled memory for cluster systems. Albeit the system offers various APIs to acess this memory such as treating it as a block device, it does not allow to expose it as a parallel file system that offers POSIX or MPI-IO semantics.**

In this paper, we 1) describe the XPD-MPIIO-driver which supports the scale out architecture of the XPDs. This MPI-agnostic driver enables high-level libraries (HDF5, NetCDF) to utilize the XPD's memory as storage. 2) A thorough performance evaluation of the XPD is conducted using IOR, this includes scale-out testing of the infrastructure and „metadata" operations, and, finally, by comparing performance to a state-of-the art Lustre system. We show that the driver and storage architecture is able to saturate wire-speed of Infiniband (60+ GiB/s with 14 FDR links) while providing low latency and little performance variability.

I. Introduction

The dilemma of the conventional high-performance storage systems is that they must maximize the bandwidth to reduce application runtimes and at the same time they shall minimize the available bandwidth to reduce costs. The first requirement is often prioritized to the detriment of the second one, which typically ends up in the oversizing and in a low average usage of the expensive bandwidth. The prioritization is motivated by the large performance peaks, that often occur in large-scale applications.

Traditional parallel file systems can be deployed on SSDs instead of HDDs, increasing performance for random workloads. Typically, data is accessed via POSIX interfaces but can be accessed using MPI-IO [1]. MPI-IO is a widely accepted middleware layer for parallel I/O that relaxes the POSIX semantics and is designed for parallel I/O. In an alternative storage architecture, a burst buffer [2], [3] is placed between compute nodes and the storage. Acting as an intermediate storage tier, it's goal is to catch the I/O peaks from the compute nodes. Therefore, it provides a low latency and good bandwidth to the compute nodes, but also utilizes the backend storage by streaming data constantly at a lower bandwidth.

In-memory systems, like the Kove® XPD® [4], provide better latency, endurance and availability as flash chips. Thus, the address space of the XPD could be used to deploy an extreme fast parallel file system. Many of the current MPI-IO implementation are optimized for the conventional storage and their I/O behavior is well understood, but in-memory systems deserve a thorough analysis.

Our **contributions** are: 1) we provide an MPI-IO implementation for the XPD 2) we investigate the performance of the developed MPI-IO driver.

While the large and scale out storage provided by the XPD is valuable by itself, the driver can be considered as an intermediate step towards a burst buffer solution.

This paper is structured as follows: Section II discusses related work, then Section III and IV describe the used API and MPI-IO implementation, Section V and VI show the test setup and performance results. Finally, the paper is summarized in Section VI.

II. Related Work

For flash based SSDs many vendors offer high-performance storage solutions, for example, DDN Infinite Memory Engine (IME) [5], IBM FlashSystem [6] and Cray's DataWarp accelerator. Using comprehensive strategies to utilize flash chips concurrently, these solutions are powerful and robust to guarantee availability and durability of data for many years.

Many existing workloads can take benefit of a burst buffer as fast write-behind cache. IME, for instance, offers to serve as a burst buffer for a storage backend allowing to transparently migrate data from flash to traditional parallel file system. In [7], a user-level InfiniBand-based file system is designed as intermediate layer between compute nodes and parallel file system. With SSDs and FDR Infiniband, they achieve on one server a throughput of 2 GB/s and 3 GB/s for write and read, respectively.

The usage of DRAM for storing intermediate data is not new and ramdrives have been used in MSDOS and Linux (with tmpfs) for decades. However, offered RAM storage was used as temporary local storage and not durable and usually not accessible from remote nodes. Exporting tmpfs storage via parallel file systems is not new but without

flushing data to a backend. Wickberg and Carothers introduced the RAMDISK Storage Accelerator [8] for HPC applications. It consists of a set of dedicated nodes that offer in-memory scratch space. Jobs can use the storage to prefetch input data prior job execution or as write-behind cache to speedup I/O. A prototype with a PVFS-based RAMDISK improved performance of 2048 processes compared to GPFS (100 MB/s vs. 36 MB/s for writes). Burst-mem [9] provides a burst buffer with write-behind capabilities by extending Memcached [10]. Experiments show that the ingress performance grows up to 100 GB/s with 128 BurstMem servers. In the field of big data, in-memory data management and processing has become popular with Spark [11]. Now there are many software packages providing storage management and compute engines [12].

The Kove XPD [4] is a robust scale-out pooled memory solution that allows to aggregate multiple Infiniband links and devices into one big virtual address space that can be dynamically partitioned. Internally, the Kove provides persistency by periodically flushing memory with a SATA RAID. Due to the performance differences, the process comes with a delay, but the solution is connected to a UPS to ensure that data becomes durable in case of a power outage. While providing many interfaces, the XPD does not offer a shared storage that can be utilized from multiple nodes concurrently.

III. XPD KDSA API

The XPD KDSA API is a low-level API that allows to send and receive data using write/read calls by utilizing RDMA. Data can be transferred synchronously or asynchronously, additionally, memory can be pre-registered for use with the Infiniband HCA. Since registration of memory is time consuming, for unregistered memory regions the system may either use an internal (pre-registered) buffer and copy the user's data to the buffer, or for larger accesses it registers the memory, performs an RDMA data transfer and then unregisters the memory again.

To address a XPD volume as a virtual address space, the XPD uses a connection specifier in the form: <local_address>/<server>.<link>:<volume ID>. Multiple volumes and client or server links can be aggregated by adding them with a +, data is then striped across these volumes/links. Similar to parallel file systems, this allows to scale the number of connections with the requirements. Upon connecting to a XPD, a thread is spawned per volume to drive the I/O, flags can control its behavior. To improve latency, this thread can use spin locks to wait for requests and transfer the data or it conserves CPU time by only becoming active upon events. The latter option is chosen as default for the MPI wrapper.

IV. XPD-MPIIO-Driver

The driver is implemented as a shared library and usable with any MPI. It can be selected at startup of an application using LD_PRELOAD with the shared library. All implemented routines check the file name for the prefix "xpd:". Without the prefix, they route the accesses to the underlying MPI. Thus, files can be selectively stored on XPD volumes. The code is available as open source[1].

The file driver implements important functions utilizing the relaxed consistency semantics offered by MPI-IO: MPI_File_open, close, delete, get_position, get_size, preallocate, read_at, write_at, read_at_all, write_at_all, read, write, seek, set_size, set_view and sync. Collective read/write are calling the independent counter part. The selection is inspired by the needs of HDF5 and IOR.

The implementation comes with a few limitations: Since we do not know the memory regions, the KDSA calls for unregistered memory are used implying overhead as described above. During the open/close the Infiniband connections to the XPD's are established and destroyed. This causes additional overhead but offers the freedom to choose the volumes on a file basis. Views support derived data types only partially, yet.

Internally, the file driver uses the shared memory space provided by one or multiple XPD volumes. It records the actual file size at the beginning of this memory region but cannot grow beyond the aggregated size of the volumes. Each process tracks its view of the file size and exchanges this information upon file close or flush as needed by MPI-IO semantics. The data space is not initialized with zeros, which is an issue if files are written in a sparse format. Since for many use cases, the file is completely overwrite, this is not a show stopper. A formatting tool is contained in the repository that initializes file size (alternatively call MPI_file_delete()) or completely zeroes memory regions.

V. Test Setup

A. Testsystems

The tests with the XPD were run on Cooley, the visualization cluster of Mira on ALCF. It provided three XPD's with a total of 14 FDR connections. Each node is equipped with one FDR HCA.

To investigate the difference between XPD and other state-of-the-art HPC systems, we run several benchmarks on DKRZ's supercomputer Mistral. Mistral hosts 3000 compute nodes each equipped with an FDR interconnect and a Lustre storage system with 54 PByte capacity. The peak transfer rate of the file system we used is 450 GiB/s[2].

B. Benchmarks

As a benchmark, IOR [13] is used varying access granularity, processes-per-node, nodes, XPD connections and access pattern (random and sequential). In all cases MPI-IO with independent I/O is measured. IOR is used with a transfer size equal to the access granularity and 20 GiB of

[1] http://github.com/JulianKunkel/XPD-MPIIO-driver
[2] http://www.vi4io.org/hpsl/2016/de/dkrz/lustre02

data per XPD connection (and volume)[3]. To synchronize the measurements and capture time for open, close and I/O separately, inter-phase barriers are turned on (IOR option -g). For the Lustre benchmarks we were trying to reuse the XPD parameters wherever possible. Collective buffer was enabled for write operations smaller than 512 KiB, we configured MPI-IO to use one aggregator per node and, in all cases the number of stripes was twice as much as the number of nodes.

VI. Evaluation

The goal is to systematically investigate the scaling behavior of the Kove XPD's. The following investigations are made: 1) scaling on 14 nodes with increasing number of connections, 2) scaling clients for 14 connections, 3) variability of performance, 4) time for open/close. Additionally, a comparison to DKRZ's Lustre is made. Since the storage capacity is rather small compared to the speed of the tests, the time for open/close are investigated separately: In average, the time for open/close reduces the reported performance by 10%. However, for production runs, larger capacities are assumed, reducing this overhead. Therefore, the performance reported subsequently in this paper is reported without the open/close time.

Note that while we measured sequential and random I/O, they behave similarly due to the DRAM storage and, thus, we only report random I/O for the XPD.

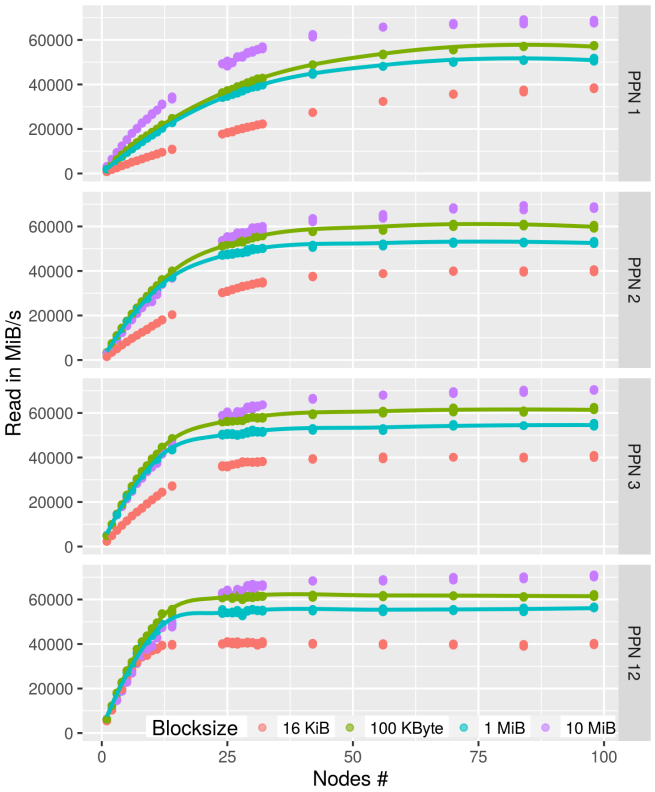
A. Scaling the number of clients

In this first experiment, the maximum number of available volumes and IB links available are used (14).

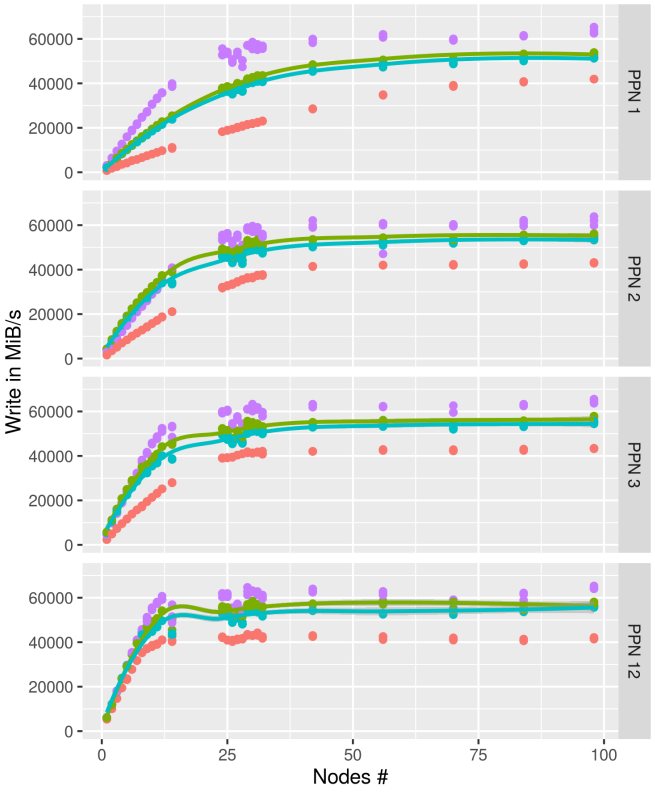
Figure 1 shows the achieved performance for 1 to 98 client nodes and 1 to 12 processes per node (performance between 3 and 12 PPN is between the measurements). Under optimal conditions, the performance should increase linearly from 1 to 14 nodes as each is equipped with one IB FDR HCA and then saturate the network.

Observations: 1) read/write behave mostly symmetrically, i.e., good read performance implies good write performance, 2) performance increases nearly with the number of client nodes and then saturates, but with PPN=1 it scales beyond 14 client odes, 3) for small access granularities, the workload is dominated by the latency of IB and the compute overhead, thus, it improves beyond 14 client nodes and using more PPN. 5) for large access granularities, a high percentage of peak is achieved quickly. Overall, 14 nodes with 12 PPN saturate at least 50% of the available network throughput and 24 clients reach almost peak. 6) performance of 100 KByte accesses is higher than for 1 MiB in many cases, this is due to the pre-registered memory region inside the KDSA library. This buffer is used for small accesses but not for 1 MiB. Therefore, the overhead for memory registration is added which slows down the I/O.

[3]The memory capacity of the XPD's is shared amongst all users, therefore, we had to deal with 20 GiB and 14 volumes.

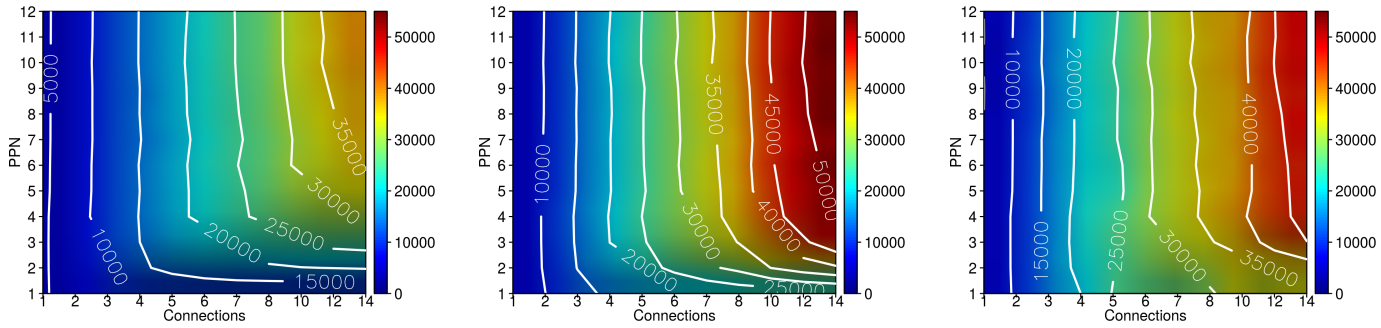


(a) Read



(b) Write

Figure 1: Performance overview: varying client node count and PPN. The graph contains fitting curves for 100 KiB and 1 MiB blocks.



(a) Granularity: 16 KiB (b) Granularity: 100 KByte (c) Granularity: 10 MiB

Figure 2: Read performance with variable connections and PPN. Isolines for multiples of 5k MiB/s are shown.

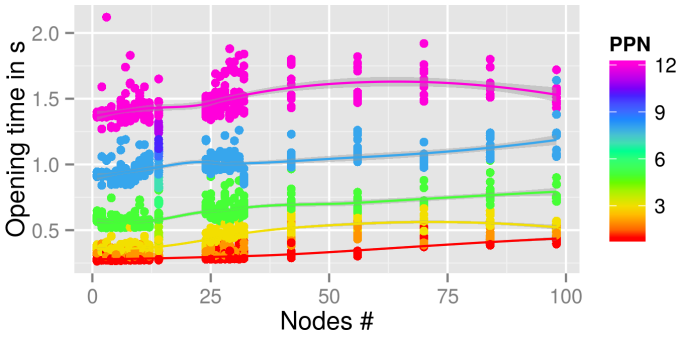


Figure 3: MPI-IO opening times for 14 XPD connections including fitting curves for PPN ∈ {1,2,3,5,8,12}

B. Scale out with multiple connections

To show the scale out behavior, the performance when varying PPN and the number of XPD connections has been measured for the fixed configuration of 14 client nodes (that should be theoretically be able to saturate all XPD connections). Figure 2 shows a heat-map for different block granularities. This gives us also another perspective to investigate scaling behavior with the PPN. In the best case, performance increases linearly with the number of connections and is constantly at a high level for variable numbers of PPN.

Observations: 1) for large accesses the performance isolines show that about 5 GB/s are achievable per connection up to 5 connections regardless of the PPN. 2) starting with 6 connections, multiple PPN are needed to drive I/O and the scaling is not optimally any more. Still, as seen in Figure 1, more PPNs and about 24 client nodes would increase throughput to 60 GiB/s. 3) smaller granularities also yield good performance with PPN=1, but the hill like structure shows that multiple PPNs are necessary to drive the latency bound I/O. Overall, the system scale well when increasing the number of XPD connections/machines.

C. Opening/closing of files

Each client process establishes its own connections to all XPDs specified in the connection string, thus, the open time is expected to depend on the number of nodes, PPN, and connections. In Figure 3, the observed time for the open (connecting) for all experiments with 14 connections is shown, close is faster. Up to the 100 nodes, the time is below 2s and 0.5s for open and close, respectively. As expected, it increases with PPN (and connections) but it does not increase much with the client nodes.

D. Comparison to Lustre

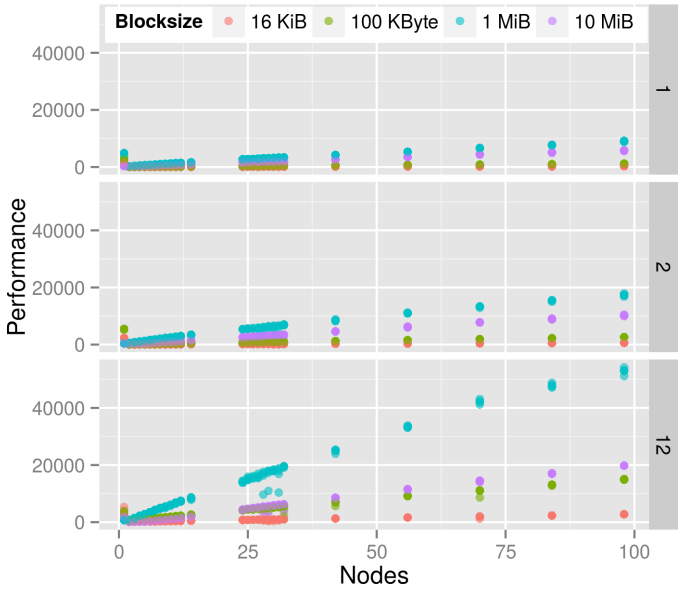
The MPI-IO performance for Lustre is shown in Figure 4. Luste uses 64 OSS and 128 OSTs in contrast to the three XPDs. Observations: read scales well with PPN and the number of nodes – except that performance drops when using two nodes instead of one. Write is similarly regardless of PPN. Reported values seem appropriate for random workloads as the currently best observed performance has been achieved during acceptance testing with sequential I/O of 2 MB blocks: with 256 nodes and PPN=16, 65k MiB/s and 122k MiB/s for write and read, respectively.

In Figure 5, the time for opening a shared file is shown. With 96 nodes and PPN=12, mean open time is 0.125 s and 0.061 s for write (i.e., create) and read, respectively this is about 1 tenth of the opening time for the XPD with 14 connections. Open time increases linearly with the number of nodes and PPN. In the same configuration, close time is lower with 0.025 s.

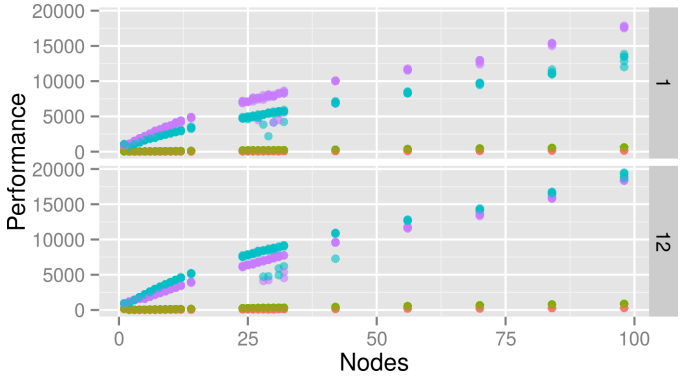
E. Performance variability

The variability of access time has been investigated. When re-running an experiment, the overall performance of a repeated run exhibits a similar performance. With the three repeats of all runs, the arithmetic mean value of runtime variability ($\frac{min-max}{max}$) is 1.23% for read and 1.78% for write accesses, albeit the mean runtime of an experiment was only about 10s.

The density (like a fine-grained histogram) of measuring timing of 10,000 individual I/Os with a single process is shown in Figure 6. The graph shows the quality difference between the Lustre on DKRZ and the XPD. As suggested by comparing application runs, the XPD's performance does not vary much between individual I/Os. While some reads in the optimized sequential I/O can perform as fast



(a) Read



(b) Write

Figure 4: Lustre performance: varying client node count and PPN. Values for other PPN are in between.

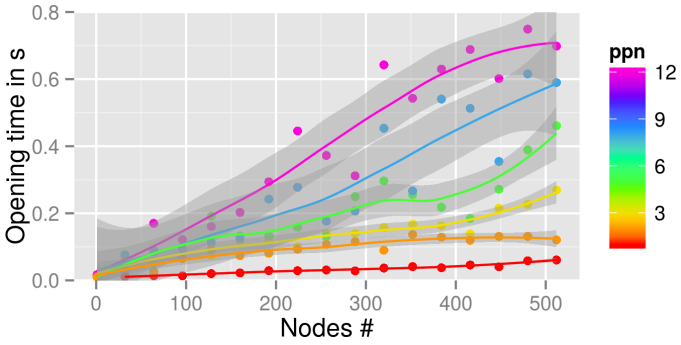
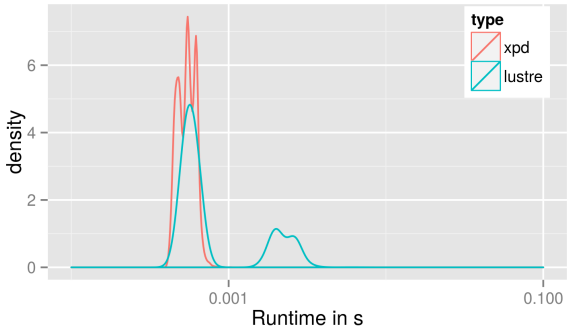
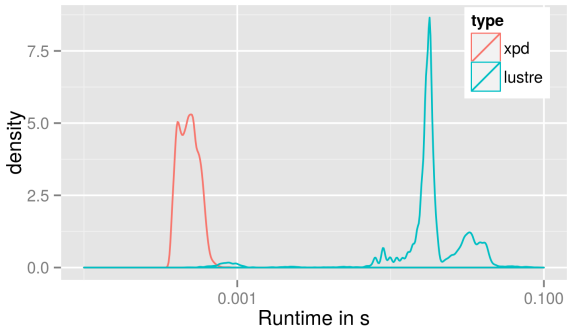


Figure 5: Lustre opening times including fitting curves for PPN ∈ {1,2,3,5,8,12}



(a) 1 MiB sequential read



(b) 1 MiB random read

Figure 6: Density of timing individual I/O operations

as on the XPD – i.e., with wire speed, most operations do not and, obviously, random I/O is significantly slower. Actually, for 16 KiB sequential reads, the read-ahead and write-behind strategy of Lustre results in slightly faster performance than the XPD (not shown).

VII. Summary

Storage on XPDs significantly outperforms our Lustre system in the small-blocks random I/O benchmarks. In this case and in contrast to XPD, the increasing number of nodes and processes don't provided the desired scaling effect. The performance benefit of the XPD is smaller when we use large granularities. From these results, it appears that this wrapper can be used to support I/O heavy workloads. In the future, we will improve the file views supporting all NetCDF scenarios, evaluate real application workloads, check the tuning options of the XPD for the MPI wrapper and provide asynchronous flushes for burst-buffer scenarios.

Acknowledgment

Thanks to Kove for their support and discussion. Thanks to our sponsor William E. Allcock for providing access and feedback. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

[1] R. Thakur, W. Gropp, and E. Lusk, "On Implementing MPI-IO Portably and with High Performance," in *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, 1999, pp. 23–32.

[2] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *In Proceedings of the 2012 IEEE Conference on Massive Data Storage*, 2012.

[3] M. Romanus, M. Parashar, and R. B. Ross, "Challenges and considerations for utilizing burst buffers in high-performance computing," *arXiv preprint arXiv:1509.05492*, 2015.

[4] *about xpress disk (xpd)*, Kove Corporation, 2015.

[5] "Worlds's most advanced application aware I/O acceleration solutions," http://www.ddn.com/products/infinite-memory-engine-ime14k, DDN, accessed: 2016-07-12.

[6] "Flash Storage," http://www-03.ibm.com/systems/storage/flash, IBM, accessed: 2016-07-12.

[7] K. Sato, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, N. Maruyama, and S. Matsuoka, "A user-level infiniband-based file system and checkpoint strategy for burst buffers," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. IEEE, 2014, pp. 21–30.

[8] T. Wickberg and C. Carothers, "The RAMDISK storage accelerator: a method of accelerating I/O performance on HPC systems using RAMDISKs," in *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2012, p. 5.

[9] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu, "BurstMem: A high-performance burst buffer system for scientific applications," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 71–79.

[10] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur *et al.*, "Memcached design on high performance RDMA capable interconnects," in *2011 International Conference on Parallel Processing*. IEEE, 2011, pp. 743–752.

[11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[12] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 7, pp. 1920–1948, 2015.

[13] W. Loewe, T. McLarty, and C. Morrone, "IOR Benchmark," 2012.